Implementando uma estratégia de redução para o contexto de Sistemas Embarcados usando CafeOBJ.

Manoel Messias Menezes, André Luis Silva, Leila Silva[1]

Departamento de Ciência da Computação e Estatística
Universidade Federal de Sergipe (UFS)
CEP 49100-000 – São Cristóvão – SE – Brasil
{manoel, andre, leila}@ufs.br

Abstract. *The focus of this work is hardware/software partitioning verification. The approach uses occam as specification and reasoning language. The partitioned system is derived from the original description of the system by applying transformation rules, all of them proved from the basic laws of occam. The aim of this work is to show how the rewriting system CafeOBJ can be used to automatically prove the partitioning rules, as well as to implement the reduction strategy that guides the application of these rules. In this way, rewriting systems can be regarded as supporting tools for the construction of partitioning environments, whose emphasis is correctness.*

Resumo. *Este artigo insere-se no âmbito da verificação formal do particionamento de sistemas em componentes de hardware e software. A abordagem adotada usa occam como linguagem de especificação e raciocínio. A descrição do sistema particionado é derivada da descrição original do sistema, mediante o emprego de regras de transformação, estas provadas usando-se as leis básicas de occam. O objetivo principal deste trabalho é explorar o uso do sistema de reescrita CafeOBJ na mecanização das provas das regras para o particionamento, bem como da estratégia de redução que gera a aplicação destas regras. Desta forma, sistemas de reescrita podem ser considerados ferramentas de suporte para a construção de um ambiente para o particionamento, cuja ênfase é o rigor formal.*

1. Introdução

Projeto integrado de hardware e software ou simplesmente *co-design* é um paradigma de projeto de sistemas que envolvem componentes de hardware e software. Nos últimos anos, várias metodologias e ferramentas para *co-design* foram propostas (por exemplo, [3, 6, 8, 15]). Um ponto central em *co-design* é a maneira pela qual o sistema é particionado nos diversos componentes de hardware e software. Neste contexto, diversas heurísticas para o particionamento foram sugeridas (por exemplo, [1, 18, 20]). Estas abordagens validam o sistema particionado por simulação. Poucos trabalhos [8, 9] usam métodos formais para provar que algumas propriedades do sistema original são preservadas após o particionamento.

Dadas a crescente diversidade e complexidade das aplicações, a mera validação do sistema após o particionamento não é suficiente para se garantir segurança. Nesse

[1] Este trabalho é parte do Projeto AbADes, financiado pelo CNPq (proc. 520071/01-8).

sentido, a verificação formal, ou seja, a prova de que o sistema particionado preserva a semântica da descrição original, faz-se imprescindível.

Em [25, 26] Silva *et al* propõem uma metodologia para o particionamento cuja ênfase é o rigor formal. Esta abordagem aceita como entrada um programa descrito em occam [13] e, a partir da descrição original, deriva a descrição do sistema particionado mediante o emprego de regras de transformação (também escritas em occam). As provas manuais destas regras, apresentadas em [25], usam as leis algébricas que definem a semântica de occam [19]. Ainda em [25] a estratégia de redução da descrição original na descrição particionada é detalhada e provada correta, usando-se indução estrutural nos construtores da gramática adotada.

Este trabalho propõe-se a complementar o rigor formal de [25] e usa o sistema de reescrita CafeOBJ [17] para provar automaticamente, bem como para implementar a estratégia de redução que guia a aplicação destas regras. Para a validação da estratégia de redução, no contexto deste trabalho foi desenvolvido um estudo de caso, cujo resultado é sumarizado nas conclusões. Na realidade, por razões de concisão, este artigo restringir-se-á somente a uma etapa da abordagem para o particionamento, a etapa de *quebra*. No entanto, o procedimento aqui ilustrado é genérico e pode ser aplicado em todo o processo do particionamento. Desta forma, sistemas de reescritas podem ser vistos como ferramentas de suporte para construção de um ambiente para o particionamento, cuja ênfase é o rigor formal. Os resultados preliminares deste trabalho, apresentados em [16], sugerem o uso do sistema de reescrita BOBJ como ferramenta de suporte à verificação em *co-design* [4]. Neste artigo, apenas provas de regras para o particionamento são exploradas. Resultados mais consolidados encontram-se em [22], onde o sistema de reescrita CafeOBJ foi explorado na completa implementação da estratégia de redução da quebra. Neste artigo, por questões de concisão, focaremos apenas nos principais resultados detalhados em [22], suficientes para ilustrar o trabalho desenvolvido.

Este artigo não pretende ser original. O objetivo aqui é relatar o trabalho desenvolvido durante o Programa de Iniciação Científica, considerando os principais resultados apresentados em [22].

Embora o trabalho aqui apresentado seja inovador no contexto de *co-design*, ressalva-se que a implementação de estratégias de redução algébrica usando sistemas de reescrita já foram propostas, por exemplo, por Sampaio [21], no contexto de verificação de compiladores (usando OBJ3 [5]) e por Lira [14], no contexto da implementação de uma estratégia de redução para linguagens orientadas a objeto (usando Maude [2]). Além disso, outros formalismos para verificação em hardware (não restritos ao particionamento) foram propostos e estão sumarizados em [12].

Este artigo está organizado como descrito a seguir. A Seção 2 descreve o formalismo utilizado neste trabalho. Em seguida, a abordagem para o particionamento é apresentada na Seção 3, onde também se detalha a fase de quebra. O sistema de reescrita CafeOBJ é descrito na Seção 4. Na Seção 5, são apresentados o procedimento de condução da prova automática das regras do particionamento propostas em [25], bem como a implementação das leis de occam, das regras do particionamento e da estratégia de redução que guia a aplicação das regras. Finalmente, na Seção 6, algumas considerações finais e trabalhos futuros são abordados.

2. O Formalismo Utilizado: A Linguagem occam

Silva [25] usa occam por duas razões principais. A primeira reside no fato de que occam é um poderoso modelo para expressar concorrência e paralelismo, fundamental para expressar a comunicação entre os componentes de hardware e software. A segunda razão deve-se ao fato de que occam é regida por um conjunto de leis algébricas [19], que definem a sua semântica. Estas leis servem como axiomas para condução das provas das regras propostas para o particionamento.

Na realidade, na abordagem proposta em [25] adota-se um subconjunto de occam, definido a seguir, usando a sintaxe expressa no estilo BNF. Por conveniência, algumas vezes a sintaxe de occam é apresentada linearmente. Por exemplo, pode-se escrever SEQ(P_1, P_2, ..., P_n) ao invés de se utilizar o estilo vertical padrão.

P ::= SKIP | STOP | x := e | ch ? x | ch ! e | IF(b_1 P_1, b_2 P_2, …, b_n P_n)
| ALT(b_1 & g_1 P_1, b_2 & g_2 P_2, …, b_n & g_n P_n) | SEQ(P_1, P_2, …, P_n)
| PAR(P_1, P_2, …, P_n) | VAR x: P | CHAN ch: P

A seguir descrevemos brevemente estes comandos (para mais detalhes veja, por exemplo, [13]). O comando SKIP não tem efeito semântico, comportando-se como um comando nulo e STOP é o processo canônico para expressar *deadlock*, não podendo realizar qualquer progresso. Os comandos ch?x, ch!e e x:=e são os comandos de entrada, saída e atribuição, respectivamente; a comunicação em occam é síncrona. Os comandos IF e ALT selecionam um processo a executar, baseados numa condição (IF) ou numa guarda (ALT). A escolha do IF é determinística, ou seja, a condição de índice mais baixo que se tornar verdadeira executa o processo a ela correspondente; mais de uma condição pode ser verdadeira num dado momento. Já o comportamento do ALT é não determinístico, ativando, aleatoriamente, o processo correspondente a uma das guardas que for satisfeita. Enquanto a condição de um IF é sempre uma expressão booleana, a guarda de um ALT envolve tipicamente comandos de entrada. Os comandos SEQ e PAR representam a composição de programas em seqüência e em paralelo, respectivamente. Os comandos VAR e CHAN denotam declarações de variáveis e canais, respectivamente. Neste trabalho é omitida a especificação de tipos de variáveis e canais, tornando implícita a validade de qualquer tipo nestas declarações. Apesar da abordagem não lidar com laços arbitrários, replicadores são permitidos nos comandos IF, ALT, PAR e SEQ, possibilitando o tratamento de vetores de processos. Por razões de concisão, replicadores não são tratados neste artigo. Assim, omitimos sua representação na descrição da sintaxe, bem como na descrição da Seção 3.1.

Na abordagem adotada para o particionamento, o subconjunto de occam foi estendido para incluir novos construtores, cuja finalidade é servir como anotações para guiar a estratégia de aplicação das regras. Estes construtores não têm nenhum efeito semântico. O construtor BOX indica que um conjunto de processos pode ser considerado um *processo atômico* no processo do particionamento. O construtor CON sinaliza que um processo não pertence à descrição original e foi introduzido pela estratégia do particionamento. Os construtores PARhw e PARsw indicam a implementação do processo em hardware e software, respectivamente. Os construtores PARser e PARpar indicam o modo de execução dos processos, se em série ou em paralelo, respectivamente.

Como mencionado anteriormente, a semântica de occam é regida por um conjunto de leis algébricas. Neste trabalho, são apresentadas apenas as leis necessárias para compreensão da estratégia descrita na Seção 3. Cada uma destas leis possui um nome, que indica seu uso, além de um número. A justificativa operacional de cada lei foi extraída de [19]**.**

As primeiras leis expressam a identidade para a composição seqüencial e paralela.

Lei 1 *(SEQ-SKIP unit)* `SEQ(SKIP, P) = SEQ(P, SKIP) = P`.

Lei 2 (*PAR-SKIP unit)* `PAR(SKIP, P) = PAR(P , SKIP) = P`.

O operador `SEQ` executa um certo número de processos em seqüência. Se não tem argumentos, ele simplesmente termina. Caso contrário, executa o primeiro argumento até terminá-lo e então executa o restante em seqüência. Portanto, o operador `SEQ` obedece a primeira lei a seguir. Já o operador `PAR` é associativo e comutativo e este fato é capturado pelas leis 4 (*PAR assoc*) e 5 (*PAR sym*), respectivamente.

Lei 3 *(SEQ assoc)* $\texttt{SEQ}(P_1, P_2, \ldots, P_n) = \texttt{SEQ}(P_1, \texttt{SEQ}(P_2, \ldots, P_n))$.

Lei 4 (*PAR assoc*) $\texttt{PAR}(P_1, P_2, \ldots, P_n) = \texttt{PAR}(P_1, \texttt{PAR}(P_2, \ldots, P_n))$.

Lei 5 (*PAR sym*) $\texttt{PAR}(P_1, P_2) = \texttt{PAR}(P_2, P_1)$.

Uma condicional `C` é ou uma expressão booleana seguida de um processo ou um construtor `IF`. A Lei 6 (*IF assoc*) é aplicada para desaninhar construtores `IFs`, tal que todos os argumentos sejam expressões booleanas seguidas de processo. Similarmente tem-se uma lei para o construtor `ALT`.

Lei 6 *(IF assoc)* $\texttt{IF}(C_1, \texttt{IF}(C_2), C_3) = \texttt{IF}(C_1, C_2, C_3)$.

Lei 7 (*ALT assoc*) $\texttt{ALT}(\texttt{ALT}(G_1), G_2) = \texttt{ALT}(G_1, G_2)$.

Quando `P` não termina imediatamente, o comportamento inicial de `SEQ(P,Q)` é aquele definido por `P`. Desta forma, `SEQ` distribui sobre o `IF`, quando este é o primeiro processo a executar.

Lei 8 *(SEQ-IF distrib)* $\texttt{SEQ}(\texttt{IF}\,_{k=1}^{n}\, b_k\ P_k, Q) = \texttt{IF}\,_{k=1}^{n}\, b_k\ \texttt{SEQ}(P_k, Q)$.

O operador `IF` distribui à direita sobre o operador `SEQ`, considerando-se que algumas condições são satisfeitas.

Lei 9 (*SEQ – IF right distrib*) $\texttt{SEQ}(P, \texttt{IF}\,_{k=1}^{n}\, b_k\ Q_k) = \texttt{IF}\,_{k=1}^{n}\, b_k\ \texttt{SEQ}(P, Q_k)$, desde que $b_1 \vee b_2 \vee \ldots \vee b_n \equiv \texttt{TRUE}$ e nenhuma variável em qualquer b_k é alterada por `P`.

O escopo de um canal local pode ser aumentado sem nenhum efeito, caso ele não interfira em nenhum outro canal de mesmo nome.

Lei 10 (*CHAN-PAR*) `PAR((CHAN ch: P) , Q) = CHAN ch: PAR(P , Q) .`

Não importa se as variáveis são declaradas em uma lista ou separadamente. A próxima lei captura este fato.

Lei 11 (*VAR assoc*) $\texttt{VAR}\ x_1 : (\texttt{VAR}\ x_2 : (\ldots \texttt{VAR}\ x_n : P)\ldots) = \texttt{VAR}\ x_1 , x_2 , \ldots , x_n : P$.

Podemos modificar o nome de uma variável local desde que este nome não seja usado numa variável livre.

Lei 12 (*VAR rename*) `VAR x: P = VAR y: P[y/x]`, se `y` não é livre em `P`.

Os construtores introduzidos em [25], para a condução do particionamento, não têm efeito semântico. Para o construtor `BOX`, por exemplo, isto é capturado pela seguinte lei.

Lei 13 *(BOX unit)* `BOX(P) = P`.

3. Abordagem para o Particionamento

A abordagem proposta em [25] para o particionamento do sistema em componentes de hardware e software deriva a descrição do sistema particionado mediante a aplicação de regras de transformação específicas para o particionamento e de leis de occam. O processo geral pode então ser resumido por:

Sistema original
= < *regras para o particionamento, leis de occam*>
Sistema particionado

Na realidade este processo está dividido em três fases: *quebra*, *definição de componentes* e *junção*.

O objetivo da fase de quebra é a transformação do sistema numa descrição que é um conjunto de processos paralelos obedecendo à *forma normal da quebra*.

Definição 1 (*Forma Normal de Quebra*) Uma descrição está na forma normal de quebra se ele tem a estrutura abaixo:

$\texttt{CHAN } ch_1, ch_2, \ldots, ch_m\texttt{: PAR}(P_1, P_2, \ldots, P_r)$

onde cada P_i, para $1 \leq i \leq r$, é *simples*.

Processos simples basicamente executam um processo atômico o qual pode ser primitivo (`SKIP, STOP, x:=e, ch ? x, ch ! e`) ou um conjunto de processos encapsulados por um construtor `BOX`. O detalhamento das possíveis formas para processos simples não é necessário no contexto deste trabalho e pode ser encontrado em [24, 25, 26].

A vantagem dessa transformação é isolar todos os processos relevantes para a análise sobre o modo de implementação, se em hardware ou em software, a ser realizado na próxima etapa. Como todos os processos estão num mesmo nível da descrição (imediatamente sob o construtor `PAR` externo) e como o operador `PAR` é associativo e comutativo, a forma normal da quebra permite flexibilidade no agrupamento dos processos que irão pertencer aos componentes de hardware e de software. Observe que qualquer permutação dos processos é permitida, sem alterar a semântica da descrição.

Na fase de definição de componentes a descrição não é substancialmente transformada. Nesta fase, heurísticas são aplicadas para se determinar quais processos comporão os componentes de hardware e de software e de que forma estes processos serão combinados, se em série ou paralelo. Dentre as métricas consideradas pelo algoritmo do particionamento estão a similaridade de funcionalidade, a similaridade do grau de paralelismo, o grau de concorrência e a dependência de dados. A decisão tomada nesta fase é indicada usando-se os construtores `PARhw`, `PARsw`, `PARser` e `PARpar` e associatividade e comutatividade do operador paralelo. A Figura 1

exemplifica a descrição de um sistema após esta fase. Por exemplo, os processos P_9, P_{10}, P_{15} e P_{18} pertencem ao mesmo componente de hardware (indicado por PARhw). Além disso, é requerido que no sistema particionado os processos P_{15} e P_{18} executem em paralelo (PARpar) e os outros dois em série com estes (PARser).

```
CHAN ch1, ch2, … , chm:
PAR
   PARsw
      PARser(P1, P5, P20)
   PARhw
      PARser(P9, P10, PARpar(P15, P18))
   PARhw …
   …
```

Figura 1-Um exemplo de uma descrição gerada após a fase de definição dos componentes.

A descrição final do sistema particionado em componentes de hardware e software é de fato obtida na fase de junção, mediante transformações algébricas. O objetivo desta fase é transformar a descrição gerada na quebra (e anotada na fase de definição dos componentes) numa descrição na forma normal de junção, a qual reflete o sistema particionado.

Definição 2 (*Forma Normal de Junção*) Uma descrição está na forma normal de junção se obedece à estrutura abaixo

```
CHAN ch1, ch2,..., chs:
PAR(Q1, Q2, …, Qt)
```

onde, comparando-se com a Definição 1, $s \leq m$, $t \leq r$ e cada P_i, $1 \leq i \leq r$ pertence exatamente a um Q_j, $1 \leq j \leq t$. Cada Q_k, $1 \leq k \leq t-1$ representa um componente de software ou de hardware e Q_t representa o componente de interface entre hardware e software.

Ao todo noventa regras para o particionamento foram propostas e provadas manualmente em [25]. Com o objetivo de mostrar como sistemas de reescritas (em particular, CafeOBJ) podem ser utilizados na automação das provas destas regras e na implementação da estratégia de derivação do sistema particionado, este artigo concentra-se na fase de quebra. Entretanto, o procedimento aqui ilustrado é genérico e pode ser estendido a todo o fluxo do particionamento. A seguir, detalharemos a estratégia de redução à forma normal de quebra.

3.1 A Estratégia de Quebra

Como mencionado, o objetivo da fase de quebra é transformar a descrição original numa descrição contendo um conjunto de processos paralelos, obedecendo à forma normal dada na Definição 1.

Para atingir a forma normal, uma estratégia de redução é aplicada, mediante a aplicação de regras de transformação. Esta estratégia compreende dois passos principais. No primeiro passo, os comandos IF e ALT são reduzidos para a forma

simples. No segundo passo, a descrição intermediária é paralelizada. Ao todo são necessárias oito regras para se reduzir um programa arbitrário (segundo a gramática expressa na Seção 2) em uma descrição na forma normal da quebra.

Por razões de concisão, neste artigo detalharemos apenas três regras do primeiro passo. As demais podem ser encontradas em [24, 25]. A Regra 1 (*IF fragmentation*), transforma um condicional arbitrário em uma seqüência de condicionais binários, um para cada ramo da condicional inicial. Esta transformação isola processos de ramos distintos do condicional, possibilitando uma análise mais flexível destes durante a fase de definição de componentes.

Regra 1 (*IF fragmentation*)

$IF_{k=1}^{n}\ b_k\ P_k$

=

$VAR_{k=1}^{n}\ c_k$:

$SEQ(BOX(SEQ_{k=1}^{n}\ c_k$:=FALSE),

$IF_{k=1}^{n}\ b_k\ c_k$:= TRUE,

$SEQ_{k=1}^{n}$ IF($c_k\ P_k$, TRUE SKIP))

desde que cada c_k é uma variável nova (ocorrendo somente onde mostrado).

Regra 2 (*ALT Fragmentation*)

$ALT_{k=1}^{n}\ (b_k \& g_k\ P_k)$

=

$VAR_{k=1}^{n}\ c_k$:

$SEQ(BOX(SEQ_{k=1}^{n}\ c_k$:=FALSE),

$ALT_{k=1}^{n}\ (b_k \& g_K\ c_k$:= TRUE),

$SEQ_{k=1}^{n}$ IF($c_k\ P_k$, TRUE SKIP))

desde que cada c_k é uma variável nova (ocorrendo somente onde mostrado).

A função do primeiro operador IF, no lado direito da Regra 1 (*IF fragmentation*), é fazer a escolha e permitir a disposição seqüencial dos comandos condicionais subseqüentes. É por isso que as variáveis novas $c_1, c_2, \ldots, c_n$ são introduzidas. Caso contrário, a execução de um dos P_i poderia interferir em algum c_j, $j > i$. Observe que o comportamento do lado esquerdo e direito da regra é o mesmo. No lado esquerdo, o primeiro b_i verdadeiro ativa o processo P_i correspondente. No lado direito, se b_i é verdadeiro, c_i é verdadeiro e os demais c_k, $k \neq i$, são falsos. Assim, somente P_i irá executar.

A separação dos ramos do ALT obedece a uma regra similar, Regra 2 (*ALT fragmentation*). A diferença desta regra para a anterior reside na substituição do processo condicional de múltiplos ramos por um processo ALT equivalente (no lado esquerdo e direito da regra). Observe que por ser BOX um processo atômico, o primeiro processo do lado direito destas regras é simples. Por outro lado, os comandos IF e ALT de múltiplos ramos executam apenas um processo primitivo (uma atribuição booleana) e, portanto, também são simples. Assim, a aplicação destas regras unifica o tratamento dos comandos ALT e IF, já que somente os condicionais binários possivelmente ainda não estão na forma simples, no caso de P_k não ser atômico. Neste caso, faz-se necessário a introdução de regras de distribuição do IF sobre operadores SEQ, PAR e IF, no restante deste texto referenciadas como *regras de distribuição do IF*. O objetivo destas regras é mover todos os condicionais para o nível mais interno da descrição. A Regra 3 (*IF SEQ distrib*), por exemplo, é usada para distribuir o IF sobre o operador SEQ.

Regra 3 (*IF SEQ distrib*)

IF(b VAR x: $SEQ_{k=1}^{n}\ P_k$, TRUE SKIP)

=

VAR c: SEQ(c := b, VAR x: $\mathrm{SEQ}_{k=1}^{n}$ IF(c P_k, TRUE SKIP))

desde que c é uma variável nova.

Após a redução dos comandos IF e ALT, a descrição do sistema consiste numa hierarquia de construtores PAR e SEQ e no segundo passo, estes construtores são tratados para a obtenção da forma normal da quebra. Assim, inicialmente a descrição é transformada em uma descrição binária pela aplicação das leis 1 (*SEQ SKIP unit*), 2 (*PAR SKIP unit*), 3 (*SEQ assoc*) e 4 (*PAR assoc*). A Lei 12 (*VAR rename*) também pode ser aplicada para tornar os nomes das variáveis locais disjuntos dos nomes das variáveis globais. Em seguida, os comandos SEQ e PAR são reduzidos.

Após este procedimento, são aplicadas as leis 4 (*PAR assoc*) e 10 (*CHAN- PAR*), visando atingir a forma normal.

Cada uma dessas regras foi provada a partir das leis básicas de occam em [25]. Para a condução desta prova inicia-se no lado direito da equação e mediante aplicação das leis de occam, obtém-se o lado esquerdo. Por razões de concisão neste trabalho mostraremos apenas o passo inicial da prova desta regra, já no contexto da aplicação do sistema de reescrita.

A estratégia da quebra pode então ser sumarizada pelo algoritmo descrito a seguir

Algoritmo 1 (*Estratégia de Quebra*)

(1) Aplique exaustivamente as leis 6 (*IF assoc*), 7 (*ALT assoc*) e 11 (*VAR assoc*).

(2) Aplique exaustivamente as regras 1 (*IF fragmentation*), 2 (*ALT fragmentation*) e as *regras de distribuição do IF*.

(3) Aplique exaustivamente as leis 1 (*SEQ-SKIP unit*), 2 (*SEQ assoc*), 3 (*PAR-SKIP unit*) e 4 (*PAR assoc*).

(4) Aplique a Lei 12 (*VAR rename*), se necessário.

(5) Aplique exaustivamente as *regras do segundo passo*.

(6) Aplique exaustivamente as leis 4 (*PAR assoc*) e 10 (*CHAN-PAR*).

4. CafeOBJ

CafeOBJ [17] faz parte de uma nova geração de linguagens de especificação e verificação algébrica, e é sucessora da família OBJ (OBJ1, OBJ2, OBJ3) [5].

A principal entidade de CafeOBJ são os módulos (*module*), os quais são compostos por tipos, operadores e equações. A sintaxe do módulo é dada por module <ModId> {}, onde <*ModId*> é o símbolo metasintático para um identificador de módulo.

Um módulo pode ser composto por três elementos. O primeiro deles, imports, especifica quais módulos devem ser importados, ou seja, herdados. Existem três formas de importação: protecting (o módulo importado não pode sofrer alteração), extending (o módulo importado pode ser estendido, porém a descrição original permanece inalterada) e using (o módulo importado pode ser estendido ou ter a descrição original modificada). O segundo elemento, signature, declara operadores, tipos e subtipos utilizados pelo módulo. Finalmente, axioms compreende variáveis e equações, as quais refletem o comportamento do módulo. Como ilustração considere o exemplo dado a seguir. O módulo QUADINT herda operações e outros módulos

definidos em NAT e INT. Na seção signature, são definidos dois tipos Nat e Int. O símbolo < indica que Nat é um subtipo de Int. O operador quad, introduzido por op, recebe um argumento do tipo inteiro e retorna o quadrado deste argumento, que é do tipo natural. O comportamento de quad é expresso por uma equação, introduzida por eq.

```
module QUADINT{
imports {
   protecting (NAT)
   protecting (INT)}
signature {
   [Nat < Int]
   op quad : Int -> Nat }
axioms {
   var I : Int
   eq quad(I) = (I * I) . }}
```

5. Mecanização do Processo do Particionamento Usando CafeOBJ

A automação da estratégia da quebra usando CafeOBJ compreende três etapas principais: a implementação das leis de occam e das regras propostas para a fase de quebra (Seção 5.1), a prova das regras propostas (Seção 5.2) e a implementação da estratégia de redução à forma normal da quebra (Seção 5.3). A seguir detalharemos cada uma destas etapas.

5.1 Implementação das Leis de occam e das Regras do Particionamento

Para implementação das leis de occam e das regras do particionamento foi necessário definir um módulo BASE, no qual são declarados tipos e operadores de uso geral. Por exemplo, foram definidos os tipos Process, Expression, List_Process, List_Expression que denotam processo, expressões, lista de processos e lista de expressões, respectivamente. Como exemplo de operadores deste módulo temos, o operador "**,**", que serve para concatenar processos, canais e variáveis, formando listas.

As leis de occam foram implementadas no módulo OCCAM-LAWS. Por razões de concisão não apresentaremos neste artigo um exemplo da implementação destas leis (ver [22] para detalhes). Entretanto, esta implementação segue o raciocínio análogo à implementação das regras mostrada a seguir.

As regras da fase de quebra foram implementadas no módulo RULES. Para esta implementação inicialmente importamos o módulo onde as leis estão implementadas, definimos a assinatura dos operadores e passamos então, na seção axioms, a implementar as equações que capturam o comportamento das regras. No exemplo a seguir, apresentamos apenas a implementação do âmago da Regra 3 (*IF-SEQ distrib*). Muitos procedimentos auxiliares não são detalhados. A descrição completa do módulo RULES encontra-se disponível em [23].

```
1  module RULES{
2  imports{protecting(BASE)}
3  signature{ ... }
4  axioms{...
5  eq gerIf(c (P , LP)) = IF(c P , true SKIP) , gerIf(c  LP) .
```

```
6  eq gerIf(c P) = IF(c P , true SKIP) .
7  eq [IF-SEQ-distrib]: IF(b (VAR LV : SEQ(LP)) , true SKIP) = VAR getV(1 (b
(LP))) : SEQ((getV(1 (b (LP))) := b)  , (VAR LV : SEQ(gerIf(getV(1 (b (LP)))
LP)))) .
8  ...}}
```

A principal equação que representa a implementação da regra em questão encontra-se na linha 7. Alguns operadores auxiliares foram utilizados. O operador `getV` é responsável pela criação da nova variável (variável `c` na descrição da Regra 3 (*IF-SEQ distrib*) da Seção 3.1). Para isso, este operador recebe como parâmetro a quantidade de variáveis que se deseja criar (no caso, uma) e a lista de processos associada a esta variável. O objetivo do fornecimento desta lista é identificar quais variáveis já estão em uso para evitar conflito de nomes. O operador `gerIf` é responsável pela criação dos processos condicionais do lado direito da Regra 3 (*IF-SEQ distrib*), na forma `IF(c` P_k`, TRUE SKIP)`. Para tal, este operador recebe como argumento a variável nova (obtida através de `getV`) e a lista de processos P_1, P_2`,...,` P_k. A implementação de `gerIf` é dada nas linhas 5 e 6. Observe que, a implementação das regras (ou leis) no sistema de reescrita não é uma tradução imediata da descrição em occam, mostrada na Seção 3.1, pois exige o uso de operadores auxiliares para se obter a transformação desejada pela aplicação das regras (ou leis).

5.2 Prova das Regras do Particionamento

Para garantir a corretude da estratégia de redução, faz-se necessária a prova das regras do particionamento propostas. A automação destas provas segue um estilo similar ao do particionamento dada em [25]. Assim, para a Regra 3 (*IF-SEQ distrib*), parte-se do lado direito e através de aplicações sucessivas das leis já implementadas, deriva-se o lado esquerdo da regra. A seguir, ilustramos apenas algumas reduções realizadas nos passos iniciais da prova desta regra, por razões de concisão. Além disso, a explicação focará apenas nos passos correspondentes à aplicação das leis de occam. Procedimentos auxiliares para a aplicação destas leis não serão detalhados.

```
1 start VAR c : SEQ((c := b) , (VAR x : SEQ(IF(c P1 , true SKIP) , IF(c P2
   , true SKIP)))) .
2 apply .seq-if-distrib at (2 1 2 2) .
3 result VAR c : SEQ((c := b) , (VAR x : IF(condicoes(c P1 , true SKIP) :
   processos(c P1 , true SKIP) : IF(c P2 , true SKIP)))) : Process
4 ...
5 result VAR c : SEQ((c := b) , (VAR x : IF(c SEQ(P1 , IF(c P2 , true
   SKIP)) , true SEQ(SKIP , IF(c P2 , true SKIP))))) : Process
6 apply .seq-if-right-distrib at (2 1 2 2 1 1 2) .
7 ...
8 result VAR c : SEQ((c := b) , (VAR x : IF(c IF(c SEQ(P1 , P2) , true
   SEQ(P1 , SKIP)) , true IF(c SEQ(SKIP , P2) , true SEQ(SKIP , SKIP))))) :
   Process
9 apply .seq-skip1 at (2 1 2 2 1 1 2 1 2 2) .
10 result VAR c : SEQ((c := b) , (VAR x : IF(c IF(c SEQ(P1 , P2) , true P1)
   , true IF(c SEQ(SKIP , P2) , true SEQ(SKIP , SKIP))))) : Process
11 apply .seq-skip2 at (2 1 2 2 1 2 2 1 1 2) .
```

Após carregar o arquivo que contém os módulos `OCCAM-LAWS e BASE` e de declarar as constantes necessárias para a condução da prova, a prova inicia-se com fornecimento do lado direito da Regra 3 (*IF-SEQ distrib*) através do comando `start`

(linha 1). Aplica-se então a Lei 8 (*SEQ-IF distrib*) (linha 2), que já está implementado no módulo occam. Os parâmetros (`2 1 2 2`) são utilizados para capturar o termo sobre o qual será aplicado a transformação, neste caso, o construtor `SEQ` mais interno. Este construtor pertence ao segundo argumento do operador `VAR` mais externo (o primeiro é a variável `c`). Este argumento, que também é um construtor `SEQ`, possui, por sua vez, um único argumento: a lista de processos expressos usando o operador "`,`". O termo a ser reduzido pertence ao segundo processo desta lista e é o segundo argumento do operador `VAR` mais interno. Após a aplicação da Lei 8 (*SEQ-IF distrib*) e a redução dos operadores auxiliares (como `processo` e `condicoes`) necessários à implementação desta lei, obtemos a descrição dada na linha 5. Em seguida, aplica-se a Lei 9 (*SEQ-IF right distrib*) (linha 6) e reduções auxiliares, obtendo-se uma descrição no qual os operadores `IF's` encontram-se distribuídos à direita sobre o operador `SEQ` (linha 8). O restante da prova prossegue analogamente. As demais regras são provadas de forma similar.

5.3 Implementação da Estratégia de Redução da Fase de Quebra

Para implementar a estratégia usando CafeOBJ, os módulos `OCCAM-LAWS` e `RULES` foram divididos em seis módulos, cada um deles refletindo o conjunto de regras/leis empregados em cada um dos passos do Algoritmo 1 (*Estratégia de Quebra*). Estes módulos foram nomeados por `Passo-i`, $1 \leq i \leq 6$. Além destes módulos, a implementação da estratégia usa o módulo `BASE`, mencionado anteriormente. Na Figura 2, apresentamos a arquitetura global do ambiente de redução, sendo que as setas tracejadas representam a dependência entre os módulos e as contínuas refletem o fluxo de execução do Algoritmo 1 (*Estratégia de Quebra*).

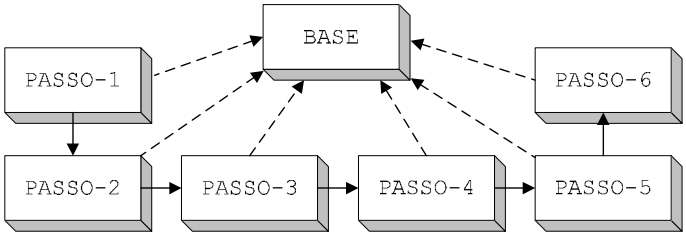

Figura 2-Arquitetura global do Ambiente de Redução .

A aplicação da estratégia dá-se da seguinte forma. Inicialmente carrega-se o módulo `PASSO-1` na memória e indicamos o termo sobre o qual a estratégia será aplicada. Em seguida, aplicamos as leis e regras contidas neste módulo e copiamos o resultado obtido o qual servirá de entrada para o módulo seguinte. A estratégia prossegue de maneira análoga até o `PASSO-6`, onde o resultado obtido será a especificação na forma normal da quebra.

6. Conclusões

Verificação formal em *co-design* encontra-se ainda num estágio incipiente e são raras as abordagens a tratar o tema. Uma das contribuições deste trabalho é complementar o rigor formal do trabalho de Silva *et al* [25, 26], no sentido de mecanizar as provas das regras e a estratégia de redução usadas na fase de quebra. Em particular, durante a automatização das provas, foram detectadas três omissões de aplicações de leis básicas nas provas manuais das regras da quebra, apresentadas em [25], atestando a importância

do uso de sistemas de reescrita como aqui proposto. Após a implementação da estratégia, para validar nosso trabalho, foi implementado o estudo de caso do programa da convolução, descrito em [26]. No sistema operacional Windows 2000 foram necessárias 250.049 reescritas e consumiu um tempo de 2 *min* e 9 *s*. No sistema operacional Red Hat Linux 9 foram necessárias 250.049 reescritas e consumiu um tempo de 14 *min* e 18 *s*

O uso de CafeOBJ se mostrou de grande valor para este trabalho. A especificação da sintaxe de occam, das leis básicas e das regras do particionamento implementadas não incorreu em grandes problemas. A implementação da estratégia de redução também foi desenvolvida naturalmente. Neste sentido, este trabalho é também uma contribuição à comunidade de CafeOBJ, pois constitui-se num estudo de caso de transformação de programas, no contexto deste sistema de reescrita. Por outro lado, para a comunidade de *co-design* este trabalho sugere que sistemas de reescrita podem ser considerados ferramentas úteis de suporte a metodologias com ênfase no rigor formal.

Apesar deste artigo se restringir apenas a fase da quebra, o procedimento utilizado na mecanização das provas e da estratégia de redução é genérico e pode ser estendido a todo o fluxo do particionamento. O próximo passo é a implementação das provas das regras da etapa de junção, bem como da estratégia de redução que guia a aplicação de tais regras, completando assim, a formalização da abordagem proposta por Silva *et al* [25]. Assim, espera-se construir um ambiente para o particionamento que deverá ser utilizado no desenvolvimento de outros estudos de caso.

Referências

[1] Barros, E. (1993) *Hardware/Software Partitioning Using UNITY*. PhD thesis, Tübingen University.

[2] Clavel, M., Durán, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J. and Quesada, J. F. (1999) *Maude: Specification and Programing in Rewriting Logic*. Computer Science Laboratory-SRI International.

[3] Ernst, R. and Henkel, J. (1992) Hardware-Software Co-design of Embedded Controllers Based on Hardware Extraction. *Handouts of the International Workshop on Hardware-Software Co-Design*, vol. de Outubro.

[4] Goguen, Joseph. *Fun with BOBJ*. Disponível on-line: http://www.cse.ucsd.edu/groups/tatami/bobj/. Último acesso 11/03/2003.

[5] Goguen, Joseph A., Winkler, T., Meseguer, J., Kokichi, F. and Jean P, J (1992). *Introducing OBJ**, Relatório Técnico. SRI-CSL-92-03, SRI International, Computer Science Laboratory.

[6] Gupta, R. and De Micheli, G. (1992) System-level Synthesis Using Re-programmable Components. *Proceedings of EDAC, IEEE Pres*, 2-7.

[7] Harper, R., MacQueen, D. and Milner, R. (1986) *Standard ML*. Edinburgh University LFCS Report Series ECS-LFCS-86-2.

[8] Hsieh, H., A. Sangiovanni-Vicentelli, F. Ballarin and L. Lavagno: 1999, Synchronous Equivalence for Embedded Systems: A Tool for Design Exploration, In *Proceedings of the International Conference on Computer-Aided Design*, pp. 505-509.

[9] Hughes, R. B. and Musgrave, G. (1996) The Lambda Approach to System Verification. Em *Hardware/Software Co-design*, Kluwer Academic Publisher, 427-451.

[10] Iyoda, J., Sampaio, A. and Silva, L (1999). ParTS: A Partitioning Transformation System. *Proceedings of FM99 (World Congress on Formal Methods)*, *Lecture Notes in Computer Science*, 1708:1400-1419.

[11] Iyoda, J. ParTS: *Um ambiente para o Particionamento de Hardware e Software* (2000), Tese de Mestrado, Universidade Federal de Pernambuco.

[12] Kern, C e M. Greenstreet. Formal Verification in Hardware Design: A Survey (1999), ACM Transactions in *Design Automation of Eletronic Systems*, 4(2):123-193.

[13] Jones, G. and Goldsmith, M. (1998) *Programing in occam 2*. Prentice-Hall.

[14] Lira, B, Cavalcanti, A. e Sampaio, A. (2002) Automation of a Normal Form Reduction Strategy for Object-Oriented Programming. *Proc. of 5th Workshop on Formal Methods*, 193-208.

[15] Madsen, J., Groge, J., Knudsen, P. V., Petersen, M. E. and Haxthausen, A. (1997) Lycos: The Lyngby Co-synthesis System. *Design Automation of Embedded Systems*, 2(2):195-235.

[16] Menezes, M., Silva, A., Silva, L (2003).Verificação Formal da Etapa do Particionamento em Co-design Usando o Sistema de Reescrita BOBJ. Anais do XXX Seminário Integrado de Hardware e Software, Campinas, São Paulo.

[17] Nakagawa.A., Sawada T. , Futatsugi, K (1999). CafeOBJ user's manual ver 1.4.2. Disponível on-line: http://www.ldl.jaist.ac.jp/cafeobj/doc/. Último acesso 01/07/2003.

[18] Niemann, R. and Marwedel, P. (1997) An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming. *Design Automation of Embedded Systems*, 2(2):165-193.

[19] Roscoe, A. W. and Hoare, C.A.R. (1988) The Laws of occam programming. *Theoretical Computer Science*, 60:177-229.

[20] Saha, D., Mitra, R. S. and Basu, A. (1997) Hardware/Software Partitioning Using Genetic Algorithm. *Proc. of 10th International Conference on VLSI Design*, India, 155-160.

[21] Sampaio, A (1997) An Algebraic Approach to Computer Design. *Volume 4 of Algebraic Methodology and Software Technology (AMAST) Series in Computing*, World Scientifc.

[22] Silva, A., Menezes, M., Silva, L. (2003). Using CafeOBJ to implement a Reduction Strategy in the Context of Hardware/Software Partitioning. Proc. of 6th Workshop on Formal Methods, pp 43-58, also in Electronic Notes in Theoretical Computer Science.

[23] Silva, L., Sampaio, A. e Jones, G. (2001) Serialising Parallel Processes in a Hardware/Software Partitioning Context, *Proceedings of the International Symposium of Formal Methods Europe*, *Lecture Notes in Computer Science*, 2021:344-363.

[24] Silva, L., Sampaio, A. and Barros, E. (1997) A Normal Form Reduction Strategy for Hardware/Software Partitioning. *Proc. of Formal Methods Europe (FME) 97*, *Lecture Notes in Computer Science*, 1313:624-643.

[25] Silva, L. (2000) *An Algebraic Approach to Hardware/Software Partitioning*. Tese de Doutorado. Universidade Federal de Pernambuco.

[26] Silva L., Sampaio A., Barros E. (to appear) A Constructive Approach to Hardware/Software Partitioning. *Journal of Formal Methods in Systems Design*.